

A Measurement and Simulation Based Methodology for Cache Performance Modeling and Tuning

Abstract

We present a cache performance modeling methodology that facilitates the tuning of uniprocessor cache performance for applications executing on shared memory multiprocessors by accurately predicting the effects of source code level modifications. Measurements on a single processor are initially used for identifying parts of code where cache utilization improvements may significantly impact the overall performance. Cache simulation based on trace-driven techniques can be carried out without gathering detailed address traces. Minimal runtime information for modeling cache performance of a selected code block includes: base virtual addresses of arrays, virtual addresses of variables, and loop bounds for that code block. Rest of the information is obtained from the source code. We show that the cache performance predictions are as reliable as those obtained through trace-driven simulations. This technique is particularly helpful to the exploration of various “what-if” scenarios regarding the cache performance impact for alternative code structures. We explain and validate this methodology using a simple matrix-matrix multiplication program. We then apply this methodology to predict and tune the cache performance of two realistic scientific applications taken from the Computational Fluid Dynamics (CFD) domain.

1 Introduction

Measurement and simulation based modeling are essentially at opposite ends of the spectrum of performance evaluation methodologies. Simulation models are typically employed for in-depth analyses of system performance under realistic operating conditions. Such models are useful for evaluating various “what-if” questions regarding system performance. However, creating and validating a simulation model is cumbersome and time consuming. In contrast, measurement-based evaluation is accurate when perturbation due to instrumentation is kept low. However, there are several limitations including: lack of repeatability, inapplicability to systems or components of a system that are not yet realized, and lack of any extensibility of current measurements to answer “what-if” performance related questions.

In this paper, we present a methodology (called “M&S”) that combines both measurement and simulation based modeling techniques for tuning cache performance for shared memory multiprocessors. This methodology retains the advantages of both techniques while avoiding their limitations. Initial measurements are carried out to locate memory-intensive portions of the code. A memory model is then constructed for a selected code block. A user may then study the cache performance impact of coding alternatives within that block. We have implemented this methodology based on a parallelizing tool, called CAPTools [14], which analyzes the Fortran77 source code and then, generates a corresponding simulation model. Initial measurements generate information about base virtual addresses of arrays and variables in selected code block and loop bounds that may not be known statically. The simulation model uses this

information to accurately predict the memory reference behavior. Thus, within a few seconds of turn-around time, a user can predict cache performance with respect to coding alternatives and select the most suitable ones for the application. We first validate this methodology using a simple matrix-matrix multiplication program. Subsequently, we model and tune the cache performance of two Computational Fluid Dynamics (CFD) applications on an SGI Origin2000, a Distributed Shared Memory (DSM) system with a cache-coherent Non Uniform Memory Access (ccNUMA) architecture.

Cache performance tuning is an important part of application development for high performance DSM systems. Without paying proper attention to multiple levels of local and remote memory hierarchies, it is very difficult to realize the performance potential of such systems. Although global address space simplifies the parallelization process, cache performance tuning remains an essential and non-trivial activity. Cache performance improvement is essentially a two-step process: the user (1) identifies code portions that exhibit poor cache performance; and (2) experiments with various coding alternatives to improve performance. Cache performance prediction in response to coding alternatives is particularly helpful at the second stage. In this paper, we limit the scope of our discussion to on-chip (primary or level one) data cache performance and single processor tuning. While memory references locality to any level of the memory hierarchy is important, our application tuning experiences reported in this paper emphasize the importance of tuning uniprocessor cache performance for scalable multiprocessor performance.

Trace-driven cache performance simulation and profiling based tuning techniques are well-known and tools based on these techniques are widely available. Extant technique or tool that try to combine these two techniques are not applicable in practice to allow a user to identify performance bottlenecks and model memory performance for detailed “what-if” analysis of possible coding alternatives aimed at improving reference locality. As we elaborate in Section 2, application of existing profiling and trace-driven simulation techniques and tools require so much time and effort that they are deemed inappropriate for cache performance tuning of real applications on existing systems. In contrast, M&S technique enables parallelization tools and compilers to parallelize a sequential application and allow code transformations for optimizing memory performance. Such techniques represent an important first step toward automating detailed memory performance analyses to an extent where they can be implemented in a parallelizing or optimizing compiler.

We distinguish our cache performance modeling methodology from other techniques with reference to related research in Section 2. Section 3 presents specific details of the M&S methodology and validates it using a simple matrix-matrix multiplication example. We then report cache performance prediction and

tuning results for two CFD applications in Section 4. Section 5 reports the impact of uniprocessor cache performance tuning on the scalability of the same programs on multiprocessors. We conclude with a discussion of this methodology and its applicability to a number of performance evaluation scenarios.

2 Related Work

Existing cache performance modeling and evaluation techniques belong to one of three broad areas: (1) analytic modeling; (2) trace- or execution-driven simulations; and (3) measurements. Analytic modeling is often used to describe the behavior of an application on a particular system under various simplifying assumptions. Analytic models are being developed for applications to predict their performance on future high-end computing systems [13]. These models can characterize the overlap of CPU and memory operations in modern processors and predict application performance on such systems [17]. Other analytic models, such as LogP [5] and LoPC [8] have been used successfully to model communication patterns for parallel applications. Analytic techniques are useful for predicting the performance of existing applications on future high performance systems. Given the cost of such systems, it is justified to invest time and resources to develop algorithm level models for these applications of critical importance. However, such efforts cannot be justified for parallelization and tuning sequential codes for high-performance parallel and distributed platforms.

A majority of existing memory subsystem modeling techniques generally target design and evaluation of processor architecture and predicting the response of real workloads on such architectures. Trace-driven simulation is a widely used technique for accurately analyzing the memory references corresponding to real workloads [4]. Various cache design alternatives are evaluated based on memory reference traces of SPEC benchmarks [9]. Trace-driven approach can be further extended to cycle-by-cycle execution-driven simulation for greater architectural details [15] or to complete system simulation for greater operating system level details [21]. All of these techniques require considerable investment on the part of a user in terms of time and effort. Even for small code blocks of real applications, the overhead of generating and handling traces is prohibitively large for accomplishing our goal of application tuning. Nevertheless, trace-driven techniques are recognized as accurate under realistic conditions for memory performance modeling [12]. Therefore, our work builds on these approaches by making them suitable for “what-if” analysis, which is essential for application source code level tuning of cache performance without requiring the user to deal with the complexities of applying the above techniques.

With on-chip support for measuring processor level activities in modern processors, measurement-based

cache performance tuning techniques are also becoming relevant [3]. Given the current state of on-chip measurement technology, it is fairly simple to make some code modifications, execute them on a real system, and use measurements from on-chip counters to analyze their impact [22]. Unfortunately, such measurements are highly dependent on system loads, therefore, the measurements are generally not repeatable. Additionally, access to these counters requires kernel level interface, therefore, their overhead is prohibitively large when they are not used for profiling the entire program. Thus, despite the potential benefits of on-chip counters, their use alone may not correctly guide the users to make code transformations that will improve cache performance. Moreover, different manufacturers provide different software interfaces to on-chip counters and the portability of the resulting code cannot be guaranteed.

A number of research efforts have tried to explore the space between the above seemingly orthogonal memory performance modeling techniques. Difference between predicted and measured cache performance has been used in *MTOOL* to detect regions of code where memory performance bottlenecks exist [11]. Use of memory management information for tuning performance has been investigated [19]. *MemSpy* uses a trace-driven simulation with profiling to investigate the causes of cache misses [18]. However, this technique cannot be applied to real applications due to its dependence on details traces to drive simulation. A number of researchers are integrating compiler level information about source code with system models at various levels of detail to evaluate and tune different aspects, including memory, performance [2,6,10]. Prefetching techniques exploit latency hiding mechanisms of modern processors to optimize memory reference locality at compile time [16]. There appears to be a trend of providing the compiler with even runtime information to allow it to play an active role in optimizing application performance statically at compile time as well as dynamically at runtime [1]. The M&S modeling methodology presented in this paper is a practical initial step toward this goal for tuning application cache performance by parallelizing tools and compilers.

3 Methodology

3.1 Memory Model

Current generation of computing systems typically involves several levels of memory hierarchy. Considering multiple levels of memory hierarchy, probability of finding a reference to memory location i (denoted by r_i) at level l is given by:

$$P[r_i \text{ is found at level } l] = p_l \prod_{j=1}^{l-1} (1 - p_j), \quad (1)$$

where p_j is the probability of finding the reference at j -th level. Since p_j is dependent on memory access characteristics of a workload, such generic models are not practical for a “what-if” analysis of memory performance due to source code level modifications. We observe that the above probability becomes a deterministic value for a given memory reference in a workload to a (possibly virtual) address on a specific architecture with multiple levels of memory hierarchy and known current states of each of those levels. Thus, given a virtual address and the current state of the memory hierarchy, we can predict whether a reference to this address will result in a hit or a miss. From a uniprocessor perspective, the state of a level l of memory hierarchy is updated whenever there is a miss in that memory level and some contents of that level are replaced with required contents from a subsequent level in the hierarchy. The selection of contents for replacement is based on a clearly defined policy, such as those *least recently used* (i.e., according to LRU policy). Since we are focusing at on-chip caches in this paper, we elaborate the model in terms of a generic set-associative cache. This cache is characterized by three parameters, as presented in Table 1.

Table 1. Parameters to characterize a cache.

Parameter	Explanation
C	Total cache capacity in bytes
B	Cache block (line) size in bytes
a	Associativity

We assume that the cache is initially in a “cold” state, therefore, initial compulsory misses should be expected. In addition, we also assume that no memory reference crosses the block boundary to simplify the calculations. The number of cache blocks (N_B) is given by: $N_B = \frac{C}{B}$, and total number of sets of cache blocks in each of the a -ways of association (N_S) can be calculated as: $N_S = \frac{N_B}{a}$. We consider a main memory where all the addresses are arranged in cache block sized segments. For instance, if A_i is the virtual address of a data structure i , the address of cache sized block of memory in which it resides is:

$$A_{b_i} = A_i - A_i \bmod(B) . \quad (2)$$

Set in cache that corresponds to the block address of reference to i is given by:

$$S_{b_i} = \frac{A_{b_i}}{B} \bmod(N_S) . \quad (3)$$

Due to a -way associativity, a reference i can be accessed from any one of a sets when it is available in cache. In case of a cache miss for reference to i , the least recently used set from a possible choices of sets will be replaced with the new block of data that also contains i . Clearly, the only unknown quantity for deterministically calculating the locality of a reference to i under the above mentioned two assumptions is the virtual address of i , i.e., A_i . We shall revisit the problem of calculating correct virtual addresses to all

Routing Slip

Mail Code	Name	Action
		Approval
		Call me
		Concurrence
		File
		Information
		Investigate and Advise
		Note and Forward
		Note and Return
		Per Request
		Per Phone Conversation
		Recommendation
		See me
		Signature
		Circulate and Destroy

Judie,

Please make a copy send
to author, and Program Manager.
original send to C. Barton
MS: 241-13. Thanks!

Christine #4502 10/28/98

Name	Tel. No. (or Code) & Ext.
Code (or other designation)	Date

are discussed in the following subsections.

3.2.1 Array References

The model for references to array elements is different from a scalar reference in terms of computation of the address for modeling a memory reference. While the address of a scalar reference can be measured once, we may have to determine addresses of individual elements for an array. Since the sizes of all the elements of an array are identical, knowing the base address of an array is sufficient to calculate addresses for all the elements. Using Fortran convention of storing an array in a column-major fashion, the address of an array element $B(I_1, I_2, \dots, I_m)$ in an m -dimensional array is calculated as:

$$Address(B(I_1, I_2, \dots, I_m)) = Address(B(1, 1, \dots, 1)) + (I_1 - 1) + \sum_{j=2}^m \left(I_j \prod_{k=1}^{j-1} \dim_B(k) \right). \quad (5)$$

3.2.2 DO Loops

In the context of our study, DO loops are important because they represent a repetitive set of memory references. If some of these repetitive references are array elements, their address is calculated according to equation (5) for each iteration of the loop. Repeated accesses to a set of memory locations within a DO loop are modeled with repetitions of modeled accesses in each iteration of the loop. In order to accomplish it, loop levels, bounds and step value of the index variables should be known. This information must be obtained by static source code analysis and runtime measurements because some of these parameters may not be known before execution.

3.3 Performance Metrics

Unlike traditional predictive models for applications, our model does not attempt to estimate execution time. Accurately predicting execution time requires modeling latency hiding mechanisms provided in modern superscalar processors and operating system activities in addition to memory references. Complexity of such a detailed model makes it inapplicable for “what-if” analyses that are the goal of our study. Consequently, it is sufficient to know whether a given version of the code better utilizes the cache compared to the another for improving overall performance i.e., reduction in execution time. We shall use *cache miss ratio*, which is equal to the number of cache misses with respect to total number of memory references, as a performance metric. This performance metric is relevant to source code tuning because different implementations of same application phase may result in different number of memory references. Cache miss ratio provides a fair comparison compared to the absolute number of misses in such cases.

3.4 Implementation

Figure 1 provides an overview of our implementation of the M&S technique. We rely on an annotated parse-tree of the code created by CAPTools. Initial measurements are needed to obtain runtime information to parameterize a selected code block. An automatic model generator then uses the parse tree of the source code and measured parameters to generate a simulation model of memory references. This model is linked with a runtime library of a cache, which is parameterized for a particular target system. Executing this model provides cache miss statistics. Comparing these results for alternative code modifications, a user can determine the most suitable modification to be incorporated in the original source code.

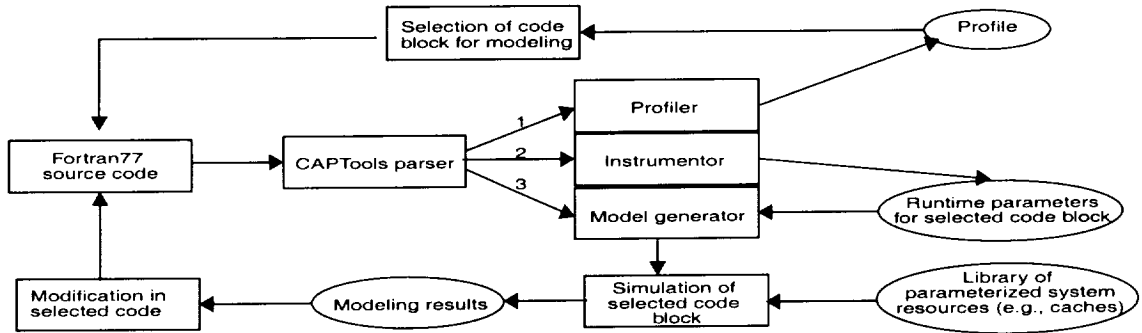


Figure 1. Implementation of M&S memory performance evaluation methodology for tuning cache utilization of Fortran77 code.

3.5 Validation

A simple matrix-matrix multiplication code is first used to illustrate and validate the use of M&S memory performance evaluation. Figure 2(a) presents a matrix-matrix multiplication implemented in Fortran77, which is written directly following the matrix-matrix multiplication algorithm without regard to the memory subsystem architecture of the target system, especially caches. Figure 2(b) represents a slightly modified version of the same algorithm with special attention to reference memory locations that are unit stride apart to ensure higher cache utilization.

By instrumenting the matrix-matrix multiplication program, we traced the base virtual addresses of each of the arrays and virtual addresses of other variables. We also log the loop bounds also as they may not be known at compile time for other codes. Figure 3 presents the trace file that results from executing the instrumented program on one processor of the Origin2000.

```

Bound: 1 64 1 0 0 0 nx 0x100133e8 4 1 1
Bound: 1 64 1 0 0 0 ny 0x100133e8 4 2 1
Bound: 1 64 1 0 0 0 nx 0x100133e8 4 3 1
Index: i 0xffff3d80 4
Index: j 0xffff3d84 4
Index: k 0xffff3d88 4
z 0xffffb0 8
x 0xffff7db0 8
y 0xffff3db0 8
  
```

Figure 3. Traced parameters of selected code block in matrix-matrix multiply program.

Figure 4 provides the simulation model for this example. This model is automatically generated using an


```

program matmul
implicit none
integer nx,ny
parameter (nx=64, ny=64)
real X(nx,ny),Y(nx,ny),Z(nx,ny)

do i = 1, nx
do j = 1, ny
do k = 1, nx
Z(i,j) = Z(i,j) + X(i,k)*Y(k,j)
end do
end do
end do
end

```

(a)

```

program matmul
implicit none
integer nx,ny
parameter (nx=64, ny=64)
real X(nx,ny),Y(nx,ny),Z(nx,ny)

do j = 1, ny
do k = 1, nx
do i = 1, nx
Z(i,j) = Z(i,j) + X(i,k)*Y(k,j)
end do
end do
end do
end

```

(b)

Figure 2. Two implementations of a matrix-matrix multiplication algorithm in Fortran77. Code in (a) shows original implementation and that in (b) shows the same algorithm with transformed loop nest. annotated parse-tree created by CAPTools. Base addresses of index variables and arrays are obtained from the initial trace and rest of the information is obtained from the parse-tree.

<pre> #include<stdio.h> int cacheSimInit(); int cacheSimL1(long long int refAddress, int refSize, char * refType, char * refVarName); void cacheSimPrintL1Stats(); int main(int argc, char ** argv) { /* definitions of index variables */ int I; int J; int K; /* index variable addresses */ long long add_I = 0xffff3d80; long long add_J = 0xffff3d84; long long add_K = 0xffff3d88; /* index variable ref sizes */ int size_I = 4; int size_J = 4; int size_K = 4; /* definitions of array addresses */ long long add_Z = 0xffffb00; long long add_X = 0xffff700; long long add_Y = 0xffff300; /* definitions of array dimensions */ int dim_Z[2] = {64, 64}; int dim_X[2] = {64, 64}; int dim_Y[2] = {64, 64}; /* definitions of array ref sizes */ int size_Z = 8; int size_X = 8; int size_Y = 8; </pre>	<pre> /* Simulation begins here */ cacheSimInit(R10K); for(I=1;I<=64;I++) { cacheSimL1(add_I, size_I, "W", "I"); for(J=1;J<=64;J++) { cacheSimL1(add_J, size_J, "W", "J"); for(K=1;K<=64;K++) { cacheSimL1(add_K, size_K, "W", "K"); cacheSimL1(add_Z + (((J))*dim_Z[0]) + ((I-1))*size_Z, size_Z, "R", "Z"); cacheSimL1(add_X + (((K))*dim_X[0]) + ((I-1))*size_X, size_X, "R", "X"); cacheSimL1(add_Y + (((J))*dim_Y[0]) + ((K-1))*size_Y, size_Y, "R", "Y"); cacheSimL1(add_Z + (((J))*dim_Z[0]) + ((I-1))*size_Z, size_Z, "W", "Z"); } } } cacheSimPrintL1Stats(); return 0; } </pre>
--	---

Figure 4. Automatically generated simulation model for matrix-matrix multiply program for evaluating primary cache misses.

The original code can be transformed to make it more cache friendly. There are at least three possible modifications, which are listed in Table Table 3. These modifications are conveniently modeled as the code generator registers the order in which different levels of loop are encountered. Since array padding changes the base addresses of various arrays, new base addresses are traced where code transformation also involves modifying array dimensions.

Table 3. Transformations of matrix-matrix multiplication code and their explanations.

Program version	Explanation
1	<i>Original matrix-matrix multiply code (see Figure 2(a))</i>
2	<i>Transformation of loop nest to make i and j change fastest and slowest, respectively, to improve locality of Z(i,j) references (see Figure 2(b))</i>
3	<i>Padding the dimensions of arrays X, Y, and Z to avoid cache conflicts due to power-of-two dimensions</i>
4	<i>Combination of 2 and 3</i>

Simulation models corresponding to four versions of the matrix-matrix multiplication example are executed and validated using two approaches. First approach emphasizes more on the functionality and is based on generating traces of each reference. These traces are compared with the memory reference traces obtained by actually running the program. The two sets of memory reference traces are completely identical, which verifies the algorithm that calculates virtual addresses of array elements relative to the base addresses. We use these traces with Dinero trace-driven simulation tool [7]. Cache miss statistics predicted by Dinero and our simulator are identical. For the second validation approach, we compare the profiled statistics obtained through *Perfex* tool on an Origin2000 with the simulator generated statistics [22]. Although *perfex* generated statistics are not accurate and depend on sampling rate of the hardware performance counter, a close match between the two quantities reassures the accuracy of our simulation methodology. Table 4 provides a comparison of primary cache miss statistics obtained from simulations as well as profiling. Note that the total number of references are calculated from simulations after carefully considering the effect of compiler optimizations on memory references. Since many compilers use registers to store array indices needed within a loop instead of issuing a memory reference every time they need to be accessed, we use this as a default for generating simulation models.

4 Cache Performance Prediction and Tuning: Two Case Studies

In this section, we present two CFD applications where the original code is incrementally modified to enable improved cache utilization on an Origin2000. Original and modified codes are modeled using the M&S methodology to predict cache performance. For both case studies, initial measurements are accomplished through instrumenting all subroutines of these applications. After selecting code blocks for

Table 4. Primary cache miss statistics for various versions of matrix-matrix multiplication programs using alternative techniques for validation.

Program version	Total number of references	Primary data cache misses			Primary data cache miss ratios (%)		
		Dinero	Simulator	Mmt.s	Dinero	Simulator	Mmt.s
1	1,581,184	12,229	12,229	14,803	0.77	0.77	0.94
2	1,581,184	3,828	3,828	6,087	0.24	0.24	0.38
3	1,581,184	9,649	9,649	12,099	0.61	0.61	0.77
4	1,581,184	1,641	1,641	3,697	0.10	0.10	0.23

cache performance modeling, further instrumentation is inserted to trace the base addresses of arrays and variables and loop bounds for the block. This instrumented version of the code executes on a single Origin2000 processor. An automatic model generator uses the parse-tree to generate cache performance models for alternative implementations of selected code blocks. These code blocks are also separately profiled using *Perfex* tool to measure cache miss statistics for comparison with predicted values.

4.1 ARC3D

ARC3D is a CFD application based on scalar pentagonal solution of Navier-Stokes partial differential equations. We apply the M&S cache performance modeling technique to tune its performance. The original code is written for vector computers and our objective is to port it to an SGI Origin2000, a DSM system. Due to global address space and loop-level parallelization extensions of the native Fortran77 compiler, shared memory parallelization of the original code is not a difficult task. However, due to differences in memory hierarchy organization, extensive cache performance tuning effort is expected. This effort starts from tuning single processor memory accesses to optimizing multiprocessor data locality.

Figure 5 shows the call graph and profile for ARC3D executed on one Origin2000 processor. Based on this profile, it is clear that the solver part of the application, RHS, requires further tuning. This part of the application solves in three spatial dimensions corresponding to the three phases of this subroutine. Therefore, we focus only on the *x*-direction solver code (to be referred to as RHSX) as the same analysis is applicable to other directions.

Figure 6 presents the Fortran77 code adapted from RHSX phase of the selected subroutine. The complexity of the code prohibits the use of any traditional cache performance modeling techniques due to their longer turn-around time. Because measurements from on-chip performance counters are almost always unrepeatable, it is difficult to use them to determine that cache performance variations are in fact due to source code modifications. We, therefore, apply the M&S technique to predict the cache performance in response to five modifications to the code segment presented in Figure 6. A detailed discussion of these

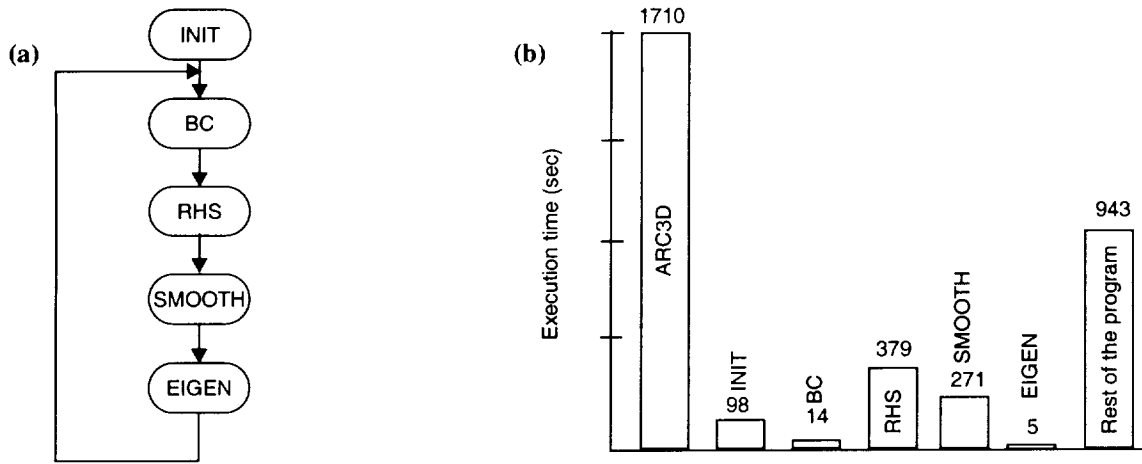


Figure 5. Original version of ARC3D. (a) Call graph and (b) profile on a single Origin2000 processor.

<pre> real xxx(64,64,64), xxy(64,64,64), xxz(64,64,64) real e(64,64,8), s(64,64,64,5), q(64,64,64,6) real qsx, pp, qsinf, pinfj, gami, uinf, vinf, winf, rx0, rx4 integer j, k, l, n do k=2,64 do l=1,64 do j=1,64 qsx = rx4 + > (xxx(j,k,l)*q(j,k,l,2) + xxy(j,k,l)*q(j,k,l,3) + > xxz(j,k,l)*q(j,k,l,4))/q(j,k,l,1) pp = (q(j,k,l,2)*q(j,k,l,2)+q(j,k,l,3)*q(j,k,l,3)+ > q(j,k,l,4)*q(j,k,l,4))*0.5/q(j,k,l,1) qsinf = (rx4+xxx(j,k,l)*uinf+xy(j,k,l)*vinf+ > xxz(j,k,l)*winf)*(1.0/q(j,k,l,6)) pinfj = (1.0/q(j,k,l,6))*1.4 e(j,l,1) = q(j,k,l,1)*qsx - qsinf e(j,l,2) = q(j,k,l,2)*qsx + xxx(j,k,l)*pp - > uinf*qsinf - xxx(j,k,l)*pinfj e(j,l,3) = q(j,k,l,3)*qsx + xxy(j,k,l)*pp - > vinf*qsinf - xxy(j,k,l)*pinfj e(j,l,4) = q(j,k,l,4)*qsx + xxz(j,k,l)*pp - > winf*qsinf - xxz(j,k,l)*pinfj e(j,l,5) = (q(j,k,l,5)+pp)*qsx - qsinf enddo enddo </pre>	<pre> do n=1,5 do j=2,64 s(j,k,2,n) = (e(j,3,n)-e(j,1,n))*(-0.5) s(j,k,64,n) = (e(j,64,n)-e(j,63,n)) enddo enddo do n=1,5 do l=3,62 do j=2,63 s(j,k,l,n) = e(j,l+2,n)+e(j,l+1,n)+ > e(j,l-1,n)+e(j,l-2,n) enddo enddo enddo enddo end </pre>
---	--

Figure 6. A code segment adapted from RHS solver in x-direction in ARC3D application.

alternative modifications is presented in the following subsections. Finally, we select the best of these modifications based on predicted cache performance improvements and combine them in the tuned implementation of the original code.

4.1.1 Array Padding

Since cache line sizes are often equal to a power-of-two value, array dimensions which are also a power-of-two values cause unnecessary conflicts to occupy same cache lines. Although a set-associative architecture

reduces the contention due to multiple sets, the severity of the problem remains significant for larger applications, such as ARC3D, due to a large number of memory accesses to an equally large number of arrays. A commonly used technique to solve this problem is to pad the arrays to increase their dimensions by one [20]. Figure 7 shows the padded arrays, while the rest of the code remains unchanged.

```
real xxx(65,65,65), xxy(65,65,65), xxz(65,65,65)
real e(65,65,9), s(65,65,65,5), q(65,65,65,6)
```

Figure 7. Array padding for RHSX phase of ARC3D.

4.1.2 Array Restructuring

Minimization of strides of array references is perhaps one of the most important technique for improving cache utilization. Unfortunately, there is no particular method of minimizing strides that can be applied in general to any given code. We try restructuring the array dimensions, so that the array elements that are used one after the other are stored in contiguous memory locations. Figure 8 shows the array declarations with their dimensions modified from the original code. The rest of the code remains unchanged.

```
real xxx(64,64,64), xxy(64,64,64), xxz(64,64,64)
real e(8,64,64), s(5,64,64,64), q(6,64,64,64)
```

Figure 8. Array restructuring for RHSX phase of ARC3D.

4.1.3 Loop Nest Transformations

Loop nest transformation is another technique that can be used for stride minimization. As it will be shown by cache performance predictions that array restructuring alone adversely impacts many references, it is interesting to try to combine the array restructuring with loop nest modifications. Figure 9 presents the modified code segment with loop nest transformations and array restructuring to minimize strides.

<pre>real xxx(64,64,64), xxy(64,64,64), xxz(64,64,64) real e(8,64,64), s(5,64,64,64), q(6,64,64,64) do l=3,62 do k=2,64 do j=1,64 same code..... enddo enddo do j=2,64 do n=1,5 s(n,j,k,2) = (e(n,j,3)-e(n,j,1))*(-0.5) s(n,j,k,64) = (e(n,j,64)-e(n,j,63))*(-0.5) enddo enddo</pre>	<pre>do j=2,63 do n=1,5 s(n,j,k,l) = e(n,j,l+2)+e(n,j,l+1)+ > e(n,j,l-1)+e(n,j,l-2) enddo enddo enddo end</pre>
---	--

Figure 9. Modified loop nests with array restructuring for RHSX phase of ARC3D to minimize strides.

4.1.4 Reducing Temporary Array Sizes

Legacy scientific applications for vector supercomputers are characterized by frequent use of large local arrays to fully utilize the vector registers. However, this strategy is counter-productive on a cache-based system because accessing larger amounts of data results in excessive cache conflicts. In order to remove these temporary arrays or reduce their sizes, excessive code alterations may be needed. Given current state of parallelization tools and compilers, reducing the cache performance effect of temporary arrays requires the effort of a programmer knowledgeable about the algorithm. Parts of ARC3D were re-written such that the dimensions of array e reduced from $e(64, 64, 8)$ to $e(64, 8)$. However, this implementation requires two additional arrays $g(64, 8)$ and $f(64, 8)$. This implementation is shown in Figure 10 as a part of the final version of the code. In addition, this implementation requires modifications in the first loop nest and some additional intermediate calculations.

4.1.5 Blocking

Blocking or strip mining is a code transformation aimed at re-using the current contents of a cache line before they have to be replaced [20]. Many compiler optimizations can automatically use this technique as a part of their compilation process. However, our experience is that automatic application of this technique is in-effective and often counter-productive for real applications of even moderate complexity. An intelligent use of this technique to only those code segments where it is needed is not only effective for optimizing local cache performance but may also improve multiprocessor data locality. We implement blocking for the RHSX code segment by defining temporary variables for six elements of array q and three variables for xxx , xyy , and xxz array elements. Since, these nine references are repeated multiple times in this code, accessing them at the beginning of the rest of the computation should keep them in cache for at least the current iteration of a loop.

Using minimal measurements from original code, we generate cache simulation models for original as well as modified code blocks. Some modifications, such as array padding and dimensions reduction, require fresh measurements of base array addresses. Table 5 presents predicted and measured cache performance of five modified versions and compares it with that of original code segment from ARC3D. We observe the following from these statistics:

- Array padding results in more than 80% reduction of cache misses in the selected code segments. Since many of the arrays used in this segment are globally used, this cache performance improvement is significant for the entire program.
- The predicted performance of the array restructured version of the code is worse than the original code. While restructuring helps minimize the strides for references to array q , strides for references to arrays

Table 5. Comparison of predicted and measured cache performance of modified versions of code with that of original code segment from ARC3D.

Version of the code under test	Total number of memory references	Predicted		Measured	
		Number of cache misses	Cache miss ratio (%)	Number of primary cache misses	Cache miss ratio (%)
<i>Original</i>	23,269,491	7,899,172	33.95	9,135,664	39.26
<i>Array padded</i>	23,269,491	1,593,534	6.85	1,396,880	6.00
<i>Array restructured</i>	23,269,491	9,151,065	39.33	5,456,944	23.45
<i>Loop nest modified</i>	14,117,811	3,393,741	19.38	2,607,424	14.89
<i>With reduced array dimensions</i>	25,397,632	7,866,036	30.97	9,480,176	37.33
<i>With blocking</i>	27,914,355	4,285,030	15.35	4,705,424	16.86

e and s actually increases. Thus, any benefits due to minimizing the strides for references to q are more than offset by cache misses resulting from larger strides of references to elements of arrays e and s. The measurement-based results are counter-intuitive as they show a reduction in cache misses. This is one of the examples where relying on measurements alone for tuning the cache performance may be misleading.

- Combining array restructuring with loop nest modifications results in more than 40% improvement in cache performance compared to the original code. Modification of loop nest helps reduce the strides with restructured arrays. Restructuring arrays alone could not achieve this objective.
- Compared to the effort involved in re-writing the code to reduce array dimensions, cache performance improvement is insignificant. This is due to ignoring other improvements that can be made by combining these changes with array padding and loop nest modifications.
- Blocking results in more than 50% reduction in cache misses due to improved cache utilization.

So far in this “what-if” study of various code transformations, we have not really combined the transformations that appear to enhance performance. Finally, we combine the following modifications to get a cache performance tuned version of the RHSX phase of ARC3D: temporary array size reduction, array padding, loop nest modifications, and blocking. Figure 10 presents the modified code segment.

Table 6 compares the predicted cache performance of the final version of the code with the original code. Combining reduction in temporary array sizes with array padding, loop nest modifications, and blocking results in about 90% reduction in cache misses. This improvement is better than cache performance improvement using any of the individual modifications alone. Due to similarity of RHSY and RHSZ with RHSX, we apply same optimizations to those phases also to improve overall cache performance.

4.2 BT

BT is an application benchmark taken from the NAS Parallel Benchmark suite. It solves a block tridiagonal system of equations resulting from approximately factored, implicit, finite-difference discretization of the Navier-Stokes equations in three dimensions. Figure 11 provides a call graph of compute and memory

<pre> real xxx(65,65,65), xxy(65,65,65), xxz(65,65,65) real e(65,9), f(65,9),g(65,9) real s(65,65,65,5),q(65,65,65,7) real qsx,pp,qsinfx,pinfj, gami,uinf,vinf,winf,rx0,rx4 real e1,e2,e3,e4,e5,e6,rx1,rx2,rx3 integer j,k,l,n do l=1,64 do k=2,64 do j=1,64 e1=q(j,k,l,1) e2=q(j,k,l,2) e3=q(j,k,l,3) e4=q(j,k,l,4) e5=q(j,k,l,5) e6=q(j,k,l,6) rx1=xxx(j,k,l) rx2=xxy(j,k,l) rx3=xxz(j,k,l) e(1,j)=e1*e6 e(2,j)=e2*e6 e(3,j)=e3*e6 e(4,j)=e4*e6 e(5,j)=e5*e6 e(6,j)=1.0/e6 qsx = rx4 + (rx1*e2 + rx2*e3 + rx3*e4)/e1 pp = gami*(e5- (0.5*(e2*e2+e3*e3+e4*e4)/e1)) qsinfx = (rx4+rx1*uinf+rx2*vinf+rx3*winf)/e6 pinfj = (1.0/e6)*1.4 f(j,1) = e1*qsx - qsinfx f(j,2) = e2*qsx + rx1*pp - > uinf*qsinfx - rx1*pinfj f(j,3) = e3*qsx + rx2*pp - > vinf*qsinfx - rx2*pinfj f(j,4) = e4*qsx + rx3*pp - > winf*qsinfx - rx3*pinfj f(j,5) = (e5+pp)*qsx - qsinfx enddo </pre>	<pre> do n = 1,5 g(n,j) = (f(n,j-2)-8.0*f(n,j-1)+ > 8.0*f(n,j+1)-f(n,j+2))* rx2 - > (e(n,j-2)-4.0*e(n,j-1)+6.0*e(n,j) > -4.0*e(n,j+2))*e1 enddo do n = 1,5 g(n,2) = (f(n,3)-f(n,1))*rx0 g(n,63)=(f(n,64)-f(n,62))*rx0 g(n,2)=g(n,2)+(0.5*e(n,1)-e(n,2)+ > 0.5*e(n,3))*e(6,2) g(n,63)=g(n,63) > +(0.5*e(n,62)-e(n,63)+ > 0.5*e(n,64))*e(6,63) enddo do n = 1,5 do j = 2, 63 s(j,k,l,n) = s(j,k,l,n) + g(n,j) enddo enddo enddo enddo enddo end </pre>
--	--

Figure 10. Modified and tuned code segment adapted from RHSX phase of ARC3D application.

Table 6. Predicted and measured performance of final version of the code as compared to the original code segment of ARC3D.

Version of the code under test	Total number of memory references	Predicted		Measured	
		Number of cache misses	Cache miss ratio (%)	Number of primary cache misses	Cache miss ratio (%)
Original	23,269,491	7,899,172	33.95	9,135,664	39.26
Tuned (final)	29,897,344	1,076,901	3.6	2,802,592	9.37

intensive subroutines and their profile based on executing class A (i.e., problem size of 64³) BT on a single processor of an Origin2000.

Based on measurement results, we choose to focus on *z_solve* subroutine. This part consists of three phases

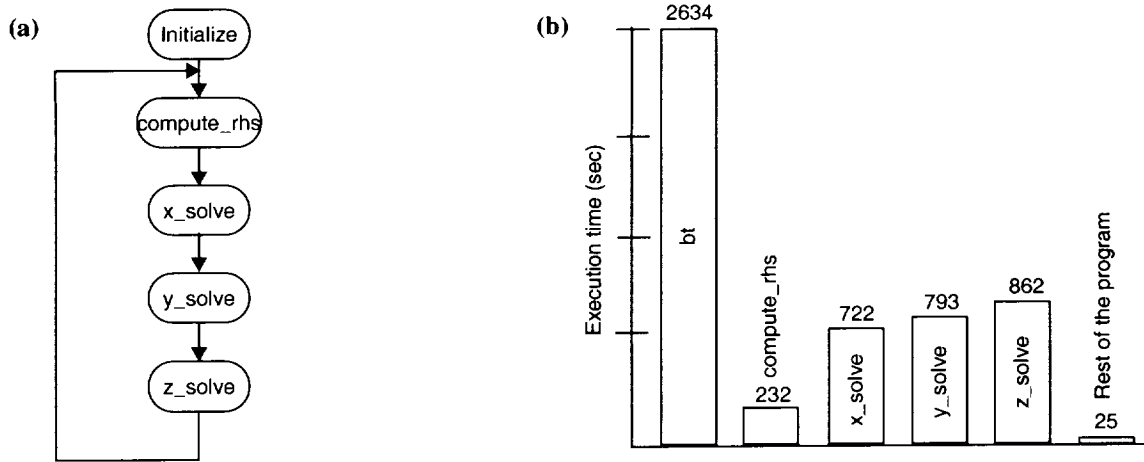


Figure 11. Original version of BT. (a) Call graph and (b) profile on a single processor of an Origin2000.

of block-tridiagonal solution of a CFD problem in z direction: calculation of left-hand-side, solution in one cell in z direction, and backsubstitution for that cell. Figure 12(a) shows the Fortran77 code of $z_backsubstitute$ phase. We opt to show this phase due to its compactness compared to other two phases. An automatically generated memory model for the $z_backsubstitute$ subroutine is used to analyze the cache miss statistics for this code, which is based on M&S technique.

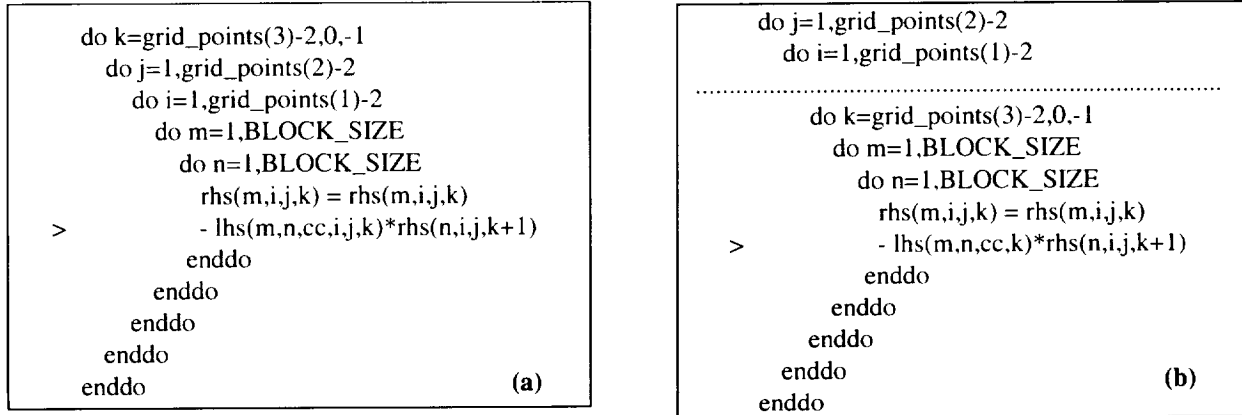


Figure 12. Backsubstitute phase $z_backsubstitute$ of BT program. (a) Original code. (b) Modified code.

A number of transformations can be applied to improve cache performance of z_solve . We reduce the dimensions of lhs array from 6 to 4. Larger temporary arrays are suitable for the original code targeted for vector systems. We also modify the loop nest to improve the reuse of recently accessed memory locations. This results in re-writing of solution algorithm in x , y , and z directions by merging the three individual phases for each. Figure 12(b) provides an overview of the transformed code related to $z_backsubstitute$.

We generate another cache simulation model corresponding to the modified code. Due to a large number of changes, we measure the base addresses of arrays corresponding to this modified version and simulate it. Table 7 compares the primary data cache misses for $z_backsubstitute$ phase from original and modified

versions of BT. Simulation predicts a reduction of cache misses by about 80% due to these modifications. Measurements based on the modified code confirm a similar level of cache performance improvement.

Table 7. Cache miss statistics from modeling and performance counter based measurements of two versions of BT on an Oririn2000 system.

Version of the code under test	Total number of memory references	Predicted		Measured	
		Number of cache misses	Cache miss ratio (%)	Number of primary cache misses	Cache miss ratio (%)
<i>Original</i>	31,974,642	2,307,653	7.22	1,793,840	5.61
<i>Tuned</i>	31,974,516	416,449	1.30	238,864	0.75

Results of these two case studies indicate that the M&S methodology yields reliable cache performance predictions corresponding to source code modifications and greatly facilitates tuning of parallelized code.

5 Multiprocessor Performance

Tuning cache performance on a single processor impacts the multiprocessor performance, especially for a DSM system, such as Origin2000. Figure 13 compares the multiprocessor performance of ARC3D and BT in terms of their original code and implementations with single processor cache tuning, discussed in Section 4. In terms of shared memory parallelization of these two programs, the original and modified versions of the two applications have no differences other than uniprocessor cache optimizations obtained in Section 4. Clearly, cache performance tuning for single processor implementation pays off in terms of almost linear scalability for multiprocessor executions as shown in Figure 13.

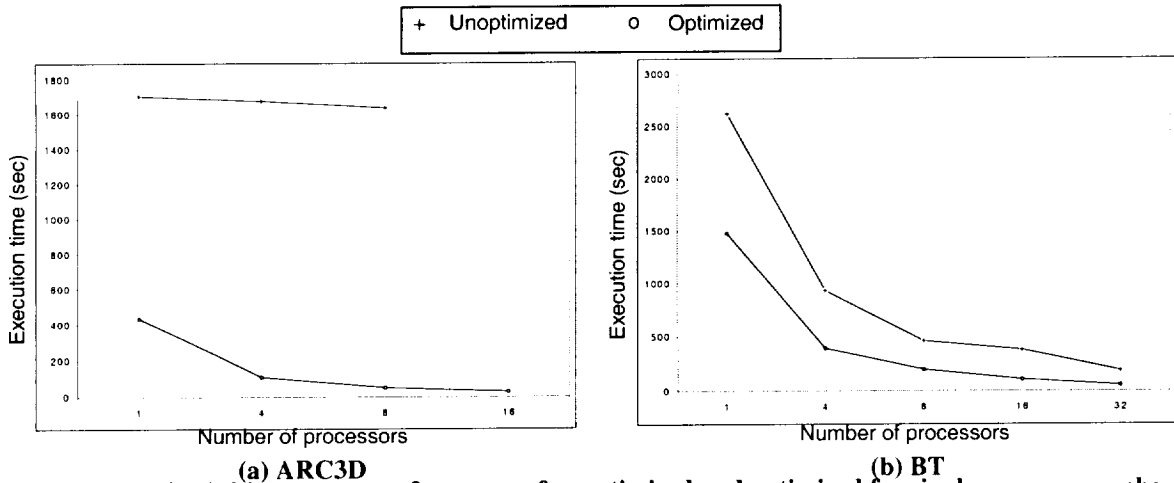


Figure 13. Multiprocessor performance of unoptimized and optimized for single processor cache utilization implementation of two applications.

6 Conclusions

We outlined a measurement and simulation based (M&S) cache performance modeling methodology and

applied it to tune the uniprocessor cache performance of two applications. M&S requires minimal trace information compared to a trace-driven simulation approach and provides extensive “what-if” analysis capabilities related to source code modifications. This capability allows us to use it in an interactive environment for parallelizing sequential code where uniprocessor cache performance plays a key role in optimizing multiprocessor performance. We underscored the importance of uniprocessor cache optimizations by comparing measurement based execution times of original and optimized versions of ARC3D and BT in Section 5. Although trace-driven simulation is a widely practiced technique in processor architecture design, it is too expensive to be applied for improving the performance of large applications. Our approach tries to couple this technique with compiler tools to make it conveniently usable by an application developer. In its current form, we have to rely on some measurement based information obtained by a single processor execution of strategically instrumented version of the code. We are investigating various techniques to get relative addresses of data structures by parsing the compiler generated object code and using a mapping function to predict the virtual addresses. While we restricted the use of M&S methodology to tune selected code blocks in this paper, M&S can easily be applied to other areas as shown Table 8.

Table 8. Scenarios of applying M&S methodology for memory performance evaluation.

Application scenario	Explanation
<i>Memory performance prediction</i>	<i>We are in the process of designing a memory performance prediction tool for Fortran77 programs and a library of system models. The main objective of this tool is to allow the user to automatically generate an extensible simulation model to study the performance of basic blocks in the code for multiple levels of memory hierarchy and different architectures.</i>
<i>Performance studies of metacomputing systems</i>	<i>Distributed high performance metacomputing systems (a.k.a., grids) may consist of heterogeneous computing resources. Simulation and measurement based techniques are useful to leverage the measurements from existing parts of the system to simulate and evaluate the parts of the system yet to be implemented.</i>
<i>Design and evaluation of memory subsystems</i>	<i>Representative workload can be used directly to simulate memory references to design and analyze architectural changes in memory subsystems by varying memory parameters. We believe that using our methodology will significantly reduce the time and effort that is currently needed for trace-driven simulations used for such design purposes.</i>

References

- [1] Sarita V. Adve et al., “Changing Interaction of Compiler and Architecture,” *IEEE Computer*, Dec. 1997, pp. 51–58.
- [2] Vikram Adve et al., “An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs,” in *the Proc. of Supercomputing '95*, San Diego, California, Dec. 1995.
- [3] G. Ammons, T. Ball, and J. Larus, “Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling,” in *Proc. of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Las Vegas, Nevada, June 1997.

- [4] P. Bose and T. M. Conte, "Performance Analysis and its Impact on Design," *IEEE Computer*, May 1998, pp. 41–49.
- [5] David Culler et al., "LogP: Towards a Realistic Model of Parallel Computation," in *Proc. of the 4th Symposium on Principles and Practices of Parallel Programming (PPoPP '93)*, May 1993, pp. 1–12.
- [6] Ewa Deelman et al., "POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems," to Appear in the *Proc. of the First International Workshop on Software and Performance*, Santa Fe, New Mexico, Oct. 1998.
- [7] J. Edler and M. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator," Available on-line from <http://www.cs.wisc.edu/~markhill/DineroIV>.
- [8] M. Frank, A. Agarwal, and M. Vernon, "LoPC: Modeling Contention in Parallel Algorithms," in *Proc. of Principles and Practices of Parallel Programming (PPoPP '97)*, Las Vegas, Nevada, June. 1997, pp. 276–287.
- [9] J. Gee, M. Hill, D. Pnevmatikatos, and A. Smith, "Cache Performance of the SPEC92 Benchmark Suite," *IEEE Micro*, August 1993.
- [10] S. Ghosh, M. Martonosi, and S. Malik, "Cache Miss Equations: An Analytical Representation of Cache Misses," in *Proc. of the 11th ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.
- [11] A. Goldberg and J. Hennessy, "MTOOL: A Method for Detecting Memory Bottlenecks," Technical Note TN-17, Digital Western Research Laboratory, Palo Alto, California, Dec. 1990.
- [12] S. Goldschmidt and J. Hennessy, "The Accuracy of Trace-Driven Simulations of Multiprocessors," in *Proc. of Sigmetrics '95*, 1995, pp. 146–157.
- [13] A. Hoisie, O. Lubeck, and H. Wasserman, "Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications," Technical Report, Los Alamos National Laboratory, Aug. 1998.
- [14] C. Ierotheou, S. Johnson, M. Cross, and P. Leggett, "Computer Aided Parallelisation Tools (CAP-Tools)—Conceptual Overview and Performance on the Parallelisation of Structured Mesh Codes," *Parallel Computing*, Vol. 22, 1996, pp. 163–195.
- [15] J. Larus, "The SPIM Simulator for the MPIS R2000/R3000," in *Computer Organization and Design—The Hardware/Software Interface* by David A. Patterson and John L. Hennessy, Morgan Kaufmann Publishers, 1994.
- [16] T. Mowry, "Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching," *ACM Transactions on Computer Systems*, 16(1), Feb. 1998, pp. 55–92.
- [17] Yong Luo et al., "Development and Validation of a Hierarchical Memory Model Incorporating CPU- and Memory-Operation Overlap," Technical Report, Los Alamos National Laboratory, Sept. 1998.
- [18] M. Martonosi, A. Gupta, and T. Anderson, "Tuning Memory Performance of Sequential and Parallel Programs," *IEEE Computer*, 28(4), April 1995, pp. 32–40.
- [19] M. Martonosi, D. Oflet, and M. Heinrich, "Integrating Performance Monitoring and Communication in Parallel Computers," in *Proc. of Sigmetrics '96*, Philadelphia, Pennsylvania, May 1996.
- [20] *Optimization and Tuning Guide for Fortran, C, and C++ — AIX Version 3.2 for RISC System/6000*, IBM, 1993.
- [21] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta, "Complete Computer Simulation: The SimOS Approach," *IEEE Parallel and Distributed Technology*, Fall 1995.
- [22] M. Zagha, B. Larson, Steve Turner, and Marty Itzkowitz, "Performance Analysis Using the Mips R10000 Performance Counters," in *Proc. of Supercomputing '96*, Pittsburgh, Pennsylvania, Nov. 1996.

How To Make A WebToon

Pamela P. Walatka

NAS-98-0XX

October 1998

walatka@nas.nasa.gov

Abstract

A WebToon is a cartoon-style table of contents for HTML documents on the web. Pictures of people are combined with word balloons that link to information within the document. The idea is to humanize the task of finding information within a document or series of documents.

Basically, to make a WebToon, you create a one-page collection of imagemaps featuring people talking. The talk is in word balloons that *link* to a relevant part of a document. The imagemaps are combined in a table that fits on one page; to be effective, the WebToons must condense the document into one page of WebToons.

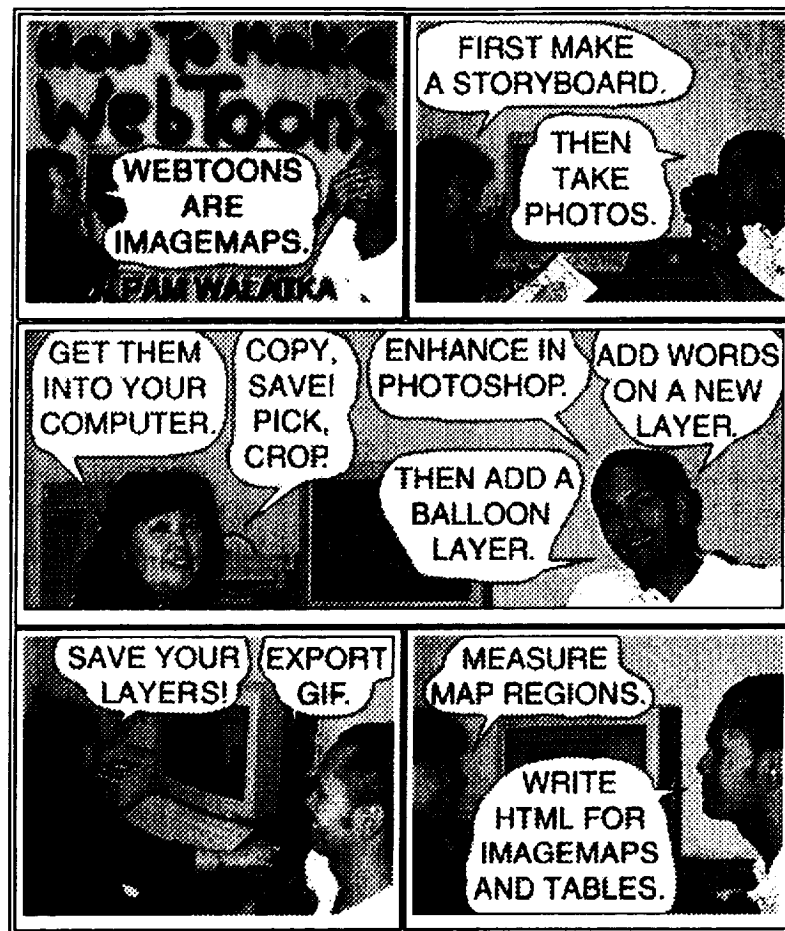
First organize your long document into chapters. Sketch a storyboard with one panel (image) for each chapter. Take photos of people acting out your storyboard. Use image processing software such as Photoshop to crop and enhance your photos. Use a new image layer to add words that briefly describe each chapter. Use another layer to put white balloons behind the words. Get the coordinates of the word balloons and then make each image into a client-side imagemap. Organize the imagemaps with an HTML table.

This report explains the details of how to make a WebToon.

See also: <http://science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-96-002/> for an example of (the first) WebToons. In the WebPrep document, click on *Glossary* for an explanation of the terms used in this document.

Table of Contents

The 'Toon About the 'Toons



In the on-line version of this report, the WebToons link to parts of the report. The user clicks on a word balloon and is taken to the relevant section.

Chapter One

WebToons Are Imagemaps

Basic Imagemap

Imagemaps are images that contain multiple links. The user clicks somewhere in the image and a new resource is retrieved: it could be another image, another page, a sound or anything else that can be represented by a URL. Just as a normal image can link to a single URL, an imagemap links to multiple URLs. You, the page creator, must map out the areas of the image that you want to relate to each specific link.

In a WebToon, each panel is a separate imagemap. This is important but not obvious. In the long run, making separate imagemaps saves work.

Here is a client-side imagemap. See the online version at <http://science.nasa.nasa.gov/people/walatka/WebToons/toon.html/chapter1.html> to see how it works.

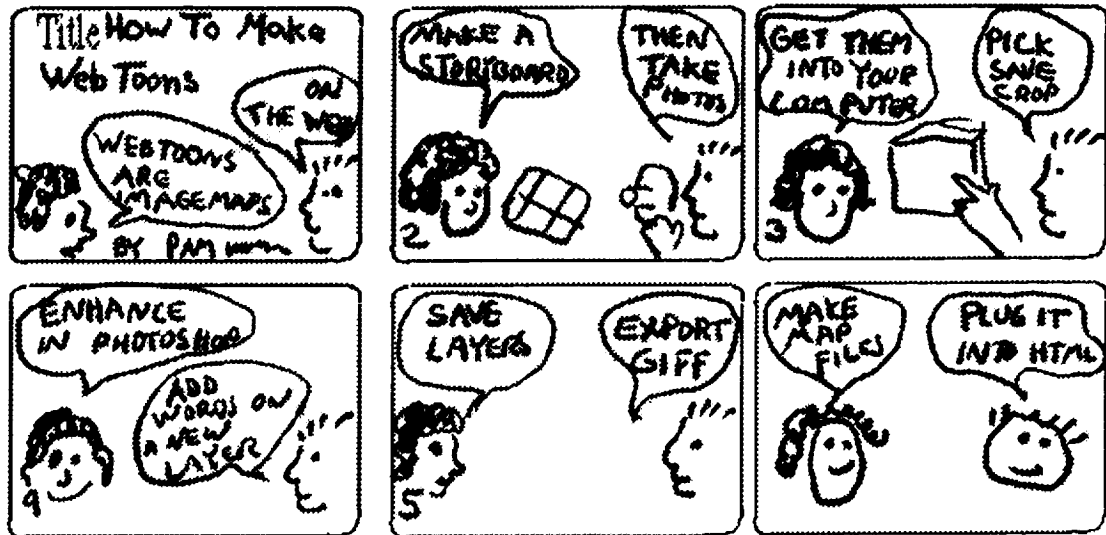


In client-side imagemaps, the list of map coordinates is embedded in the HTML document. For details, see the section on Map Coordinates in Chapter 5.

Chapter Two

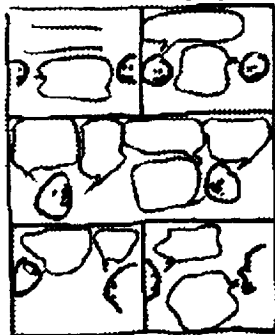
Storyboards and Photos

First Make a Storyboard



A storyboard is a **rough** sketch of how the WebToon is going to look. I use paper and pencil. Points to keep in mind:

- It all has to fit on one page. If you were to use more than one page, you would be violating the basic idea of condensing everything onto one page.
- First sketch out some boxes (panels). I think nine boxes are the maximum that fit on one page. Five is good, with the middle one long.



See Appendix A for example storyboard layouts you can use.

- Rough in some word balloons.
- Start filling in the word balloons with the major points of your document. This is **very** hard, and may require several iterations. Get a new sheet of paper and start over. Or you could make copies of the boxes and balloons and try different ways of filling in the words. If you are the author of the document, you might find yourself reorganizing the chapters to make more

sense. Each panel should represent one chapter or section. The words have to fit in the balloons; you **MUST** condense. Take some time; do it many times until you have the essence.

- If you have an idea of what the actors should be doing, sketch them in, without worrying about how good they look. For example, when I was sketching frame two of these Toons, I knew I needed Joel to be using a camera.
- You are finished with the storyboard when you have a set of simple, easy-to-understand word balloons that show the key ideas of your document, along with the positions of your actors.

Creating a WebToon Image

The next phase involves creating the WebToon digital images based on the storyboard. Now you have a style choice to make: do you want photographic images or traditional-cartoon drawings?

You could draw the images

There is no requirement for the images used in WebToons to be photographic. I originally tried drawing them in Adobe Illustrator, but the results were poor, due to my drawing ability. One of the hardest parts was getting the characters to look consistent from panel to panel. If you can draw, I encourage you to try it, electronically or on paper (and then scan). Eventually, the images must be digital in order to be made into imagemaps on the web.

Photos are fast

I found photography a quick and easy means of getting images of people.

Taking Photographs for WebToons

Before you start taking pictures, make sure you have a **COMPLETED** storyboard and you therefore know what you want your actors to do. This will expedite the filming process. For actors, use your friends and colleagues. Give your actors at least a day's warning so that they can prepare what to wear. Perhaps solid color clothing would be easier to "select" when you are working with the images, later. Use a setting with plenty of light.

- **Digital camera is fastest**

The fastest and most technically elegant way to get the digital images for WebToons is to use a digital camera of good quality. You will need a resolution of at least 400 x 300 pixels; this gives you the chance to crop down to 200 x 150. Another alternative would be to use a regular film camera and then scan. My final images are 200 x 150 at 96 dpi.

Note: be sure to test your camera, in the room that you will be using, and download a few test images to make sure that the camera settings are correct.

- **Leave room over the actors' heads**

This is hard. When you are taking the photos, imagine the word balloons over the heads. The general rule of thumb is to leave the top HALF of the image for the balloons.

- **Take twice as many snapshots as you think you need**

More is better. You can choose which ones to use later. You may have to take many shots before the actors relax. Sometimes the best shots will surprise you.

- **Use props**

Whenever possible, have something in the picture to illustrate what you are talking about. If you are talking about fire extinguishers, show the people holding up a fire extinguisher. (Later, when you make imagemaps, have significant parts of the fire extinguisher be links to relevant information.) If you are talking about something with visible-physical steps, show the steps in your images.

Chapter Three

Working With the Images

Get the images into your computer

- **From digital camera**

Many digital cameras come with an A/V line that you hook up to a computer that has the appropriate software for loading the images. Other cameras write the photos to a removable disk.

- **From film**

You can have your film developed onto a CD ROM, or get prints and then scan, at medium resolution. You want to have a least 400 pixels wide by 300 high; this gives you room to crop.

- **From drawings**

If you drew the images on paper, scan them. If you drew them electronically, they are already in your computer.

Image Processing

At this point, you are ready to start working with the images in your computer. First you will make backups, then load your files into an image processing program, pick out which images to use, and crop them to get just the important parts of the images. Then you will create layers: background, people, balloons, and words. You will save your layered files for future revisions, and save out each panel as a non-layered GIF file.

Photoshop from Adobe works well as an image processing program for WebToons. The following instructions include details about using Photoshop. NAS staff and users: Photoshop is available on the NAS SGIs; just type `photoshop` at the Irix prompt. The latest version for the SGI is 3.0.1, which these instructions are based on. A later version of Photoshop is also available on the machines in the NAS Multimedia Lab.

Pick and crop the images you want to use


- **Make extra copies**

You may mess up your images and want to go back to the originals, or need to copy a part of the image out of the original. Make and save copies of all your originals.

- **Look, list, pick**

Load your image files into an image processing program such as Photoshop. Spend some time looking at the images you have. Bring as many as possible up on your screen at the same time. Give each one a representative name, and make a list of what you have. Pick the most expressive and interesting shots, and decide which will go where. That is, pick one image for each panel of your storyboard. (A cartoon panel is one frame or box or picture-- nine or fewer panels make one page or one WebToon).

- **Crop, leaving extra space on top**

Use your image processing program to crop all the images to the same (or coordinated) size(s). In this WebToon, I used 200 x 150 pixels at 96 dpi for each panel, except for the extra-wide middle panel, which is 400 x 150. The size will depend on your storyboard; remember that you want the entire WebToon to fit on one browser screen. In Photoshop, you can set the cropping tool to repeat the same dimensions and resolution. Double-click on the Crop tool  and the Cropping Tools Options dialog box will appear. Click Fixed Target Size. Crop down to the most interesting parts of the image. Remember to leave room--the top half of the image--for the word balloons.


Add layers to your images, and enhance

If your image processing program offers layers, you can create several different layers of your image, and work separately on each layer. Working on layers allows you to make changes to one layer without messing up the pixels on the other layers. It would be possible but treacherous to make WebToons without layers.

- **Photoshop (3 or above) offers layers**

Layers make it possible, for example, to rewrite the words without changing the pixels on the other parts of the image. This is important. When you are working with layers, you put the words on a separate layer and the balloons on a separate layer to facilitate corrections.

The first thing to do is make a copy of the background layer. In Photoshop, select Window/Palettes/Show Layers to get the Layers palette. Drag the rectangle representing the Background layer down to the new-layer icon.

 Then click the eye of the Background layer off, and never look at or work on the background again, unless you need to replace messed-up pixels. The active layer is highlighted in the layer palette--click on a layer name to highlight it and make it the layer you are working on. (When you

get a message like "no-pixels-selected," you probably are not working on the layer you think you are).

Your final image will have the following layers (from the bottom up):

1. Background
2. Background copy
3. People
4. Balloons
5. Words

See the snapshot of a Photoshop session at the top of Chapter Four to see how the layer palette looks.

Remember that each panel is a separate image which will become a separate imagemap which you will assemble into a table with borders. You do not need to make borders on your images; HTML will do it for you, through the TABLE tag.

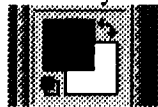
● Use layers to pop the colors (people)

When an artist draws cartoons, the artist can make the characters stand out from the background by making the characters bigger than normal, and more vividly colored. With photographs, there are ways to make the characters stand out. One way is called "popping the color." To pop the color means to isolate the subject of interest and then make the rest of the image grayscale. To do that in Photoshop:

1. Drag the Background copy layer to the new layer icon. Name the new layer People.

2. Select the people:

Check to see that the People layer is highlighted in the Layers palette, to indicate that that is the active layer. Make a very rough selection of one of the people by choosing the lasso tool, holding down Alt or Option, and clicking around the person. Make the foreground color black by clicking on the tiny black box near the color indicator boxes



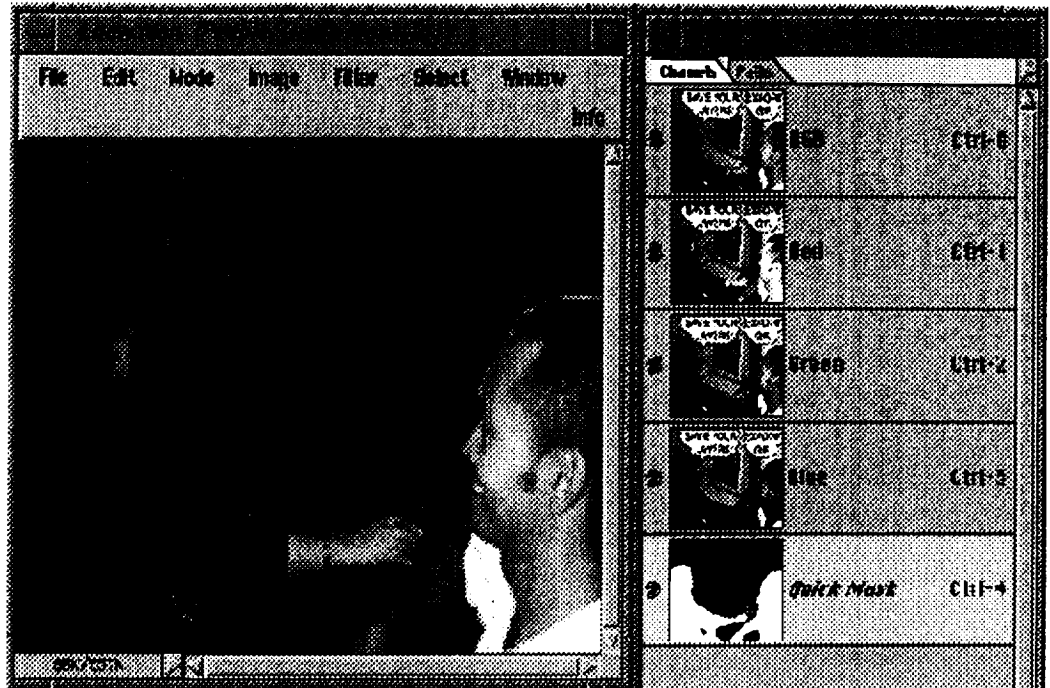
. Go into mask mode by clicking on the right-hand mask



icon. Now you are ready to use the paintbrush to refine your selection. Click on the paintbrush and paint away any areas of the background that are not under the red mask. Use the Brushes palette to change the size of your brush as necessary. If parts of the person are covered by the mask, click on the tiny arrows by the color indicator boxes to change the current color to white (Also click the tiny boxes if the background color was not white). Using white removes the mask. Keep working until you have a pretty good selection of the person. Remember, brushing with black adds to the mask; brushing with white removes the mask. The areas not under the mask are the areas that will become selected as you exit mask mode.

That is, mask mode enables you to brush in the details of any selection you are trying to make. You can adjust the edge of your selection back and forth until you get it right. When you click on the

left-side quick mask icon  the mask goes away and your unmasked areas are selected.



Photoshop Quick Mask. The red area is the mask. The not-red areas will become the selection when you exit Quick Mask mode. To add to the mask area, paint with black. To add to the selection, paint with white. Note that the palette showing on the right is the Channels palette (not Layers). You choose which palette to show, from the Windows menu.

3. Then choose Select/Save Selection. You now have a reusable selection of your person. Repeat the process with the other person or persons or characters in that image.
4. Retrieve the first selection you saved (channel 4) by choosing Select/Load Selection/OK. Repeat, changing the channel number on the Load Selection dialog box to the next one up, and clicking Add to Selection. Repeat until you have loaded all the selections you saved for this image.
5. All your people should now be selected. Choose Select/Save Selection to save the combined selections for possible future use.
6. Choose Select/Inverse to select the area around the people.
7. Hit the Delete or Backspace key to remove the background from this layer. You will not see any difference, because the background is still on the layer underneath.
8. In the layers palette, click on the Background Copy layer to make it

active. Choose Image/Adjust/Desaturate. Your people should now be popped.

Also, you can now add a halo by adding a layer between the people and the Background Copy layer. Choose a fuzzy brush from the Brushes palette, and white paint as the foreground color. Just brush at the edge of the people, with the layer between the people and the Background Copy active. Use a separate layer so that mistakes are easy to correct.

- **Adjust levels; enhance as necessary**

With the People layer active (click on it in the Layers palette) choose Image/Adjust/Levels. In the middle of the Levels dialog box, under Input Levels, you will see a histogram (graph) representing the brightness and darkness levels in your people layer. Drag the little black triangle on the left about half an inch toward the middle. Do the same with the white triangle on the right--drag toward the middle. Adjusting the levels like this makes your grayish tones more black and white, increasing punch.

Depending on your available time and talent, make other enhancements.

At this point, I would recommend that you go back and bring your other images up to this stage before adding the words and balloons. Remember that you should have at least three and not more than nine images, one for each panel of the WebToon, and that the panels will be assembled into a table. The table supplies the borders.

Add the words

Use your storyboard as a guide, but feel free to revise.

- **Use a new layer!!!!**

Do not start typing on any of your other layers. I find I need to type the words several times to get them to fit right. When they have their own layer, this is no problem. Also, I find I need to be able to revise. As time goes by, the facts change and I need to be able to change the words without messing up the background pixels. Therefore, use a separate layer just for words. Remember that you have saved copies of your originals if you accidentally type on your background. Notice that you do not have the balloons yet--they come later.

- **Be brief**

You may have to condense the words several times to get them to fit into the panel. Simplify!


- **Be legible**

Use a plain font such as Helvetica in at least 12 point size. Use all caps, as in ordinary comics. If your handwriting is great and you have a digitizing pen, you could hand letter the words.

Add balloon layer

- **New layer, *under* words**

Use a separate layer for the balloons--you will want to redraw them without messing up your words. In Photoshop, on the Layers palette, click the new

layer icon  and name the layer Balloon, then drag the Balloon layer icon between the People and Background Copy layers. That is, put the balloon layer behind the words. Use white paint, with a big hard brush (if the Brushes palette is not already showing, choose Window/Palettes/Show Brushes). To make the triangle to the mouth, hold down Alt (or Option) while clicking a triangle with the lasso. Then choose Edit/Fill/ to fill with white. To make the black line around the balloon, change the foreground color to black and then hold down Control + Alt (or Option) while clicking to select the balloons. With the balloons selected, choose Edit/Stroke and in the dialog box select one pixel, inside, foreground color. Then do Control D (or Select/None) to undo the balloon selection. If you need to revise the balloons, select the balloon layer, select the balloons with the marquee tool, delete, and brush them in again.

Chapter Four

Saving Your Files

Save the Layers in Your Files!



- You **will** need your layers
- You **will** make changes

One of the great things about the web is the ease with which you can update your documents. The need for revision arises. To facilitate future changes, you will want to be sure and retain the layers in your images. In Photoshop version 3.x and higher, this usually is not a problem; just be careful not make the mistake of flattening your image, or merging layers and then saving the file to the same filename you used for the layered image. Make a point of saving the layered files and saving backups to another machine or removable media.

Export GIF

- Photoshop/File/Export/Gif89a

CompuServe GIF is a graphic image format that works well for imagemaps and is readable on all graphics-capable web browsers. Photoshop version 3.0 and higher offers a safe and convenient method for saving GIF files.

1. Use File/Export/Gif89a. This works well and leaves your layered file unaltered. Gif89a gives you the option to interlace your gif file, which makes it appear to load faster because the user can see the gist of the image as it loads. Leave Interlace clicked on.
2. Sometimes, mysteriously, Gif89a is not available (for example, with grayscale). You can use File/Save a copy/CompuServe GIF. This method also protects your layered files, but your files will be bigger than Gif89a files and interlace is not offered.

If you are not familiar with the gif file format, look it up in the Glossary at

<http://science.nasa.gov/Pubs/TechReports/NASreports/NAS-96-002/glossary.html>

Chapter Five

Making Imagemaps and Tables for WebToons

Map Coordinates

See our introduction to imagemaps in Chapter One if you are unclear on the basic idea of imagemaps.

Essentially there are two types of imagemaps: client-side and server-side. Client-side imagemaps are the newer and preferred technology. The instructions below are for client-side imagemaps. Note: in changing my old server-side imagemap files to client-side files, I found that I needed to move the "default" line to the bottom of the map list, in order for the imagemaps to work on both major brands of browsers.

What You Need to Make an Imagemap

Once you have created your WebToon images you are ready to turn each of your images into an imagemap. Remember that in a WebToon, each panel (image) is a separate imagemap.

1. Subdivide the image into clickable regions--in WebToons, the word balloons are the clickable regions. I usually use each panel to represent one chapter. Each balloon corresponds to one section within the chapter.
2. For each region (word balloon) you will need the following, each of which are discussed in the paragraphs below:
 1. In the target document, an `<a name>` tag to mark a specific section as the target
 2. The URL to which the region should link
 3. The region coordinates.

`<A NAME>` Anchor

In order to link to a section within your HTML document, you will need to edit your HTML document and add anchor tags, with NAME attributes, at the top of the section. For example, ` ` That is, your imagemap can link to a particular section within the file if the section has an `<a name>` tag and closing `` tag, usually on a line by themselves just above the title of the section.

The URL

Each clickable region of your WebToon image will link to a specific URL (Uniform Resource Locator, web address). If your HTML files are within the

same directory as the WebToons, you only have to use relative URLs, not absolute path URLs. For example, if all your files are in the `http://science.nas.nasa.gov/~walatka/WebToons` directory or folder, your URL for Chapter Three could be just `chapter3.html` instead of the full pathname.

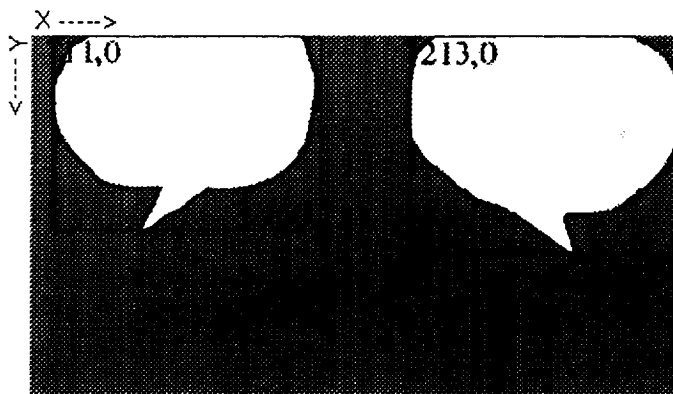
To link to a particular section within a document (as opposed to the top of the document) make use of the A NAME anchors you added to your target document. In the URL, use the value (for example, `section2`) of the A NAME anchor, preceded by a pound (#) sign. Like this: `href="chapter3.html#section2"`

Region coordinates

You need to get the pixel coordinates of your clickable region (the word balloon). Your image must be in its final dimensions. I use the UNIX utility `xv` or Photoshop to measure image coordinates. You can find automatic mapping programs on the web; see the References section for some suggestions.

Image coordinates are two-dimensional, measured in pixels. The origin (zero-zero) is at the upper-left corner of the image, with X increasing from left to right, and Y increasing from top to bottom.

The diagram below may clarify. I have sketched in temporary rectangles to roughly approximate the balloons. (The imagemap creator has the option of using rectangles, circles, or polygons. For now, let's use rectangles for simplicity, and because some browsers do not yet recognize the other shapes).



Balloon Layer, Image for Chapter 3

The upper-left corner of the first rectangle is at `x=11` and `y=0`. That is, the upper-left corner is 11 pixels over from the corner of the image, and zero pixels down from the top. The lower-right corner is the other corner necessary to define the rectangle. It is located 160 pixels over (x-direction) from the left side and 111 pixels down from the top.

The `x,y` coordinates are separated by commas, and pairs of coordinates are separated by commas. The coordinates for the first rectangle look like this:

coords="11,0,160,111"

Tip 1: Do not overlap your coordinates. If one rectangle is 0,0,200,75 then the one below it would start at 0,76 not 0,75.

Details: I determine the coordinates by looking at the image in Photoshop, and setting the cross-hair section of the Show Info palette to pixels. [If the Info palette is not showing, choose Window/Palettes/Show_Info, then click-and-hold on the cross in the x-y box at the bottom of the Info palette and choose "Pixels."] I draw (or imagine) temporary rectangles on a new layer, and put my cursor in the upper-left corner of the rectangle. Then I note the pixel coordinates in the cross-hair section of the Info palette. Next I note the coordinates of the lower-right corner. This process is repeated for each clickable region.

When you have your A NAME tags, your region coordinates and corresponding URLs, you are ready to write the HTML code for the imagemaps. (Mapping programs probably help you with this).

Imagemap HTML Code

There are three elements of an imagemap:

1. the map name
2. the list of areas (regions) and corresponding resources: see the section above, beginning at the top of this chapter for instructions on how to construct your list of areas
3. the image, in GIF format

the map name

Give the map a name.

```
<map name="3toon.map">
```

the list

For *each* of the imagemap clickable regions,

1. Specify the shape of the region. Some current browsers recognize only the rectangular shape and default, others offer these options:
 - rect for rectangular areas
 - circle for circular areas
 - poly for polygonal areas
 - default for background areas--in case the user clicks inside the imagemap but outside of the defined regions. ****Always put the default line last in your list of regions.****
2. Calculate coordinates -- see the "Region Coordinates" section above for instructions on getting the pixel coordinates. For rects, give the upper left and lower right coordinates in pixels. That is, in pixels, how far away is the upper left corner of the region from the upper left corner (origin) of the image? And how far from the origin is the lower-right corner? For circles,

give the x and y for the center point and the radius in pixels. For polygons, the format is x comma y comma for each point of the polygon, going all the way around the the polygon.

The code looks like this <area shape="poly"
coords="79,51,145,53,146,93,133,95,61,87"
href="myfile2.html#header6">

3. href -- give the URL you want to link to, in standard href format; for a file in the same directory, just give the filename. For a specific section within the file, use the filename, a pound (#) sign, and the section's <a name> tag (see the "The URL" section above).

Quiz

In the diagram in the previous section, if the first balloon were meant to link to the top of Chapter 3, the imagemap HTML code for the first balloon would look like this:

```
<area shape="rect" coords="11,0,160,111"  
href="chapter3.html">
```

See if you can figure out the code for linking the second balloon to chapter3.html, section2. The answer is in Appendix B.

The image and the end of the imagemap

In addition to giving the map name and a list of regions, you will need to call the image and close the map. See the example below.

```
<area shape="rect" coords="11,0 160,111" href="chapter3.html">  
<area shape="rect" coords="213,0 365,164"  
href="chapter3.html#section2">  
<area shape="default" href="chapter3.html">  
</map> 
```

Note the # sign before the map name.

Tip2: If your map does not work, double-check the punctuation of each set of coordinates:

x comma y comma x comma y comma.

Putting the Imagemaps Together in Tables

You could stack the imagemaps together using
 tags to arrange them in rows, but you have come this far, why not do it right and put the imagemaps into tables? With tables, you can get nice borders and good alignments. I will walk you through it.

If you are unfamiliar with the basic HTML TABLE elements, see the Table Section in WebPrep at:

science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-96-002/WebPrepChp4.html#HDR7 for introductory info.

Start with a hard copy of your storyboard--your layout plan for your WebToon. Sketch in (or merely notice) the layout of the rows. The example shown has three rows.

Top	Row
Middle Row	
Bottom	Row

In the HTML document for your WebToon page, type in the standard HTML for a table with three rows. Some HTML editing programs could do this for you--or you could borrow the sample code in the Appendix, or work through the progressive examples below. This first example is just the framework of the rows, we will fill in the cells later.

```

<!-- ***TABLE of IMAGEMAPS for WEBTOONS*** -->
<table border width="414" >
<tr> <!-- Top Row -->

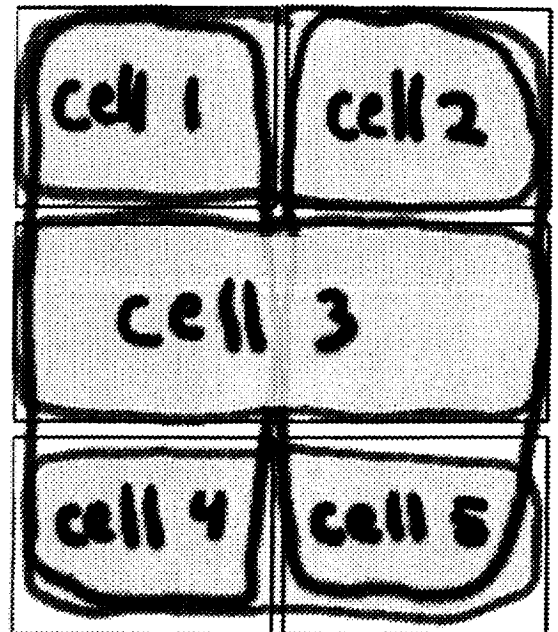
</tr> <!-- End Top Row -->
<tr> <!-- Middle Row = all one toon -->

</tr> <!-- End Middle Row -->
<tr> <!-- Bottom Row-->

</tr> <!-- End Bottom Row-->
</table>

```

When you have your rows coded in, as above, you are ready to add your table data cells (TDs). Notice in the storyboard and in the sketch at right, that this particular layout has two cells in the top row, one in the middle, and two on the bottom row. Thus, the one cell in the middle row must span two columns; you use the `colspan="2"` attribute in your TD tag.



The cell sketch

Here is the sample code with cells added, but not the imagemaps yet.

```

<!-- ***TABLE of IMAGEMAPS for WEBTOONS***-->
<table border width="414" >
<tr> <!-- Top Row-->
    <td> <!-- ***Toon 1***-->

    </td>
    <td> <!-- ***Toon 2***-->

    </td>
</tr> <!-- End Top Row-->

```



```

<tr> <!-- Middle Row = all one toon--just one cell-->
      <td colspan="2" > <!-- ***Toon 3***-->

      </td>
</tr> <!-- End Middle Row-->

<tr> <!-- Bottom Row-->
      <td> <!-- ***Toon 4***-->

      </td>
      <td> <!-- ***Toon 5***-->

      </td>
</tr> <!-- End Bottom Row-->
</table>

```

Take a look at the HTML above and get a feel for the structure of the table. When you understand how the rows and cells are arranged, you are ready to plug in your imagemap code, as follows. The indents are for clarity, not required. See Appendix C for the complete code.

```

<!-- ***TABLE of IMAGEMAPS for WEBTOONS***-->
<table border width="414" >
<tr> <!-- Top Row-->
      <td> <!-- ***Toon 1***-->
      CODE FOR IMAGEMAP 1 GOES HERE
      </td>
      <td> <!-- ***Toon 2***-->
      CODE FOR IMAGEMAP 2 GOES HERE
      </td>
</tr> <!-- End Top Row-->

<tr> <!-- Middle Row = all one toon--just one cell-->
      <td colspan="2" > <!-- ***Toon 3***-->
      CODE FOR IMAGEMAP 3 GOES HERE
      </td>
</tr> <!-- End Middle Row-->

<tr> <!-- Bottom Row-->
      <td> <!-- ***Toon 4***-->
      CODE FOR IMAGEMAP 4 GOES HERE
      </td>
      <td> <!-- ***Toon 5***-->
      CODE FOR IMAGEMAP 5 GOES HERE
      </td>
</tr> <!-- End Bottom Row-->

```

</table>

*

Summary of Imagemaps-in-Table HTML

1. Make an HTML table that corresponds to the WebToon storyboard. For each horizontal row of the storyboard, there should be a corresponding row (TR) in the table. For each panel (image) in the row, there should be a corresponding table cell (TD).
2. Place the HTML code for one imagemap within each table cell.
 - To make an imagemap, first name the map: <map name="1toon.map">
 - Then type the area shape, coords, and href for each clickable area within the toon image (each word balloon) plus an area called "default" linking to the top of the chapter, for the regions outside of your defined areas (default is optional).
 - Close the map: </map>
 - Call the image, giving width, height, alt, and USEMAP.
 - Close the TD.
3. When you are at the end of the row, close the TR.
4. When you are at the end of the Toon, close the table.

Summary of How To Make WebToons

- Write a related series of documents.
- Make a storyboard summarizing the documents.
- Create digital images of people talking.
- Copy, save, pick, crop the images.
- Enhance the images in Photoshop or other image processing software.
- Add words on new layers and balloons on new layers.
- Save layered files and GIF files.
- Create an imagemap for each GIF image.
- Assemble the imagemaps into an HTML table.
- Place the table into an HTML document that will serve as your Table of Contents. Optional: also provide a text-only Table of Contents, on another page, for those who prefer reading plain text.

If you do not get it, write to me: walatka@nas.nasa.gov. If you succeed in making a WebToon, let me know where it is. Good luck!

Acknowledgments

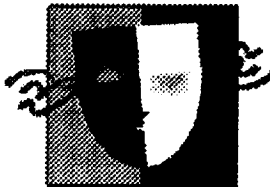
I am indebted to Sandy Johan, Val Watson and Sharon Marcacci for excellent suggestions, and to Sam Uselton for guidance, and to Randolph Kaemmerer for help with the manuscript. IHiP provided good imagemap information on the web.

Annotated References

1. An outline of this paper was published in: Walatka, Pamela P., "WebToons: A Method for Organizing and Humanizing Web Documents," Visual Proceedings, The Art and Interdisciplinary Programs of SIGGRAPH96, Computer Graphics Annual Conference Series 1996, Association for Computing Machinery, New York, ISBN 0-89791-784-7, ACM Order No. 428961, p. 156.
2. IHiP, at <http://www.ihip.com/cside.html> gives good info and links to mapping tools.

3. www.yahoo.com/Computers_and_Internet/Internet/World_Wide_Web/Imagemaps/ has links to imagemap information, including imagemap editors.
4. science.nas.nasa.gov/Pubs/TechReports/NASreports/NAS-96-002/WebPrepChp4.html is part of my guide to putting scientific reports on the web. Chapter Four is about imagemaps; the Glossary explains many of the terms used in this document.
5. The NAS Webweaver's page
science.nas.nasa.gov/Services/Education/Resources/webweavers.html
links to good resources for web workers, including guides to HTML.
6. John December and Mark Ginsburg, HTML 3.2 & CGI Unleashed, Sams.net Publishing, 1996, is a thick, complete, correct book on HTML, including tables and imagemaps. See page 401 for links to imagemap editors.

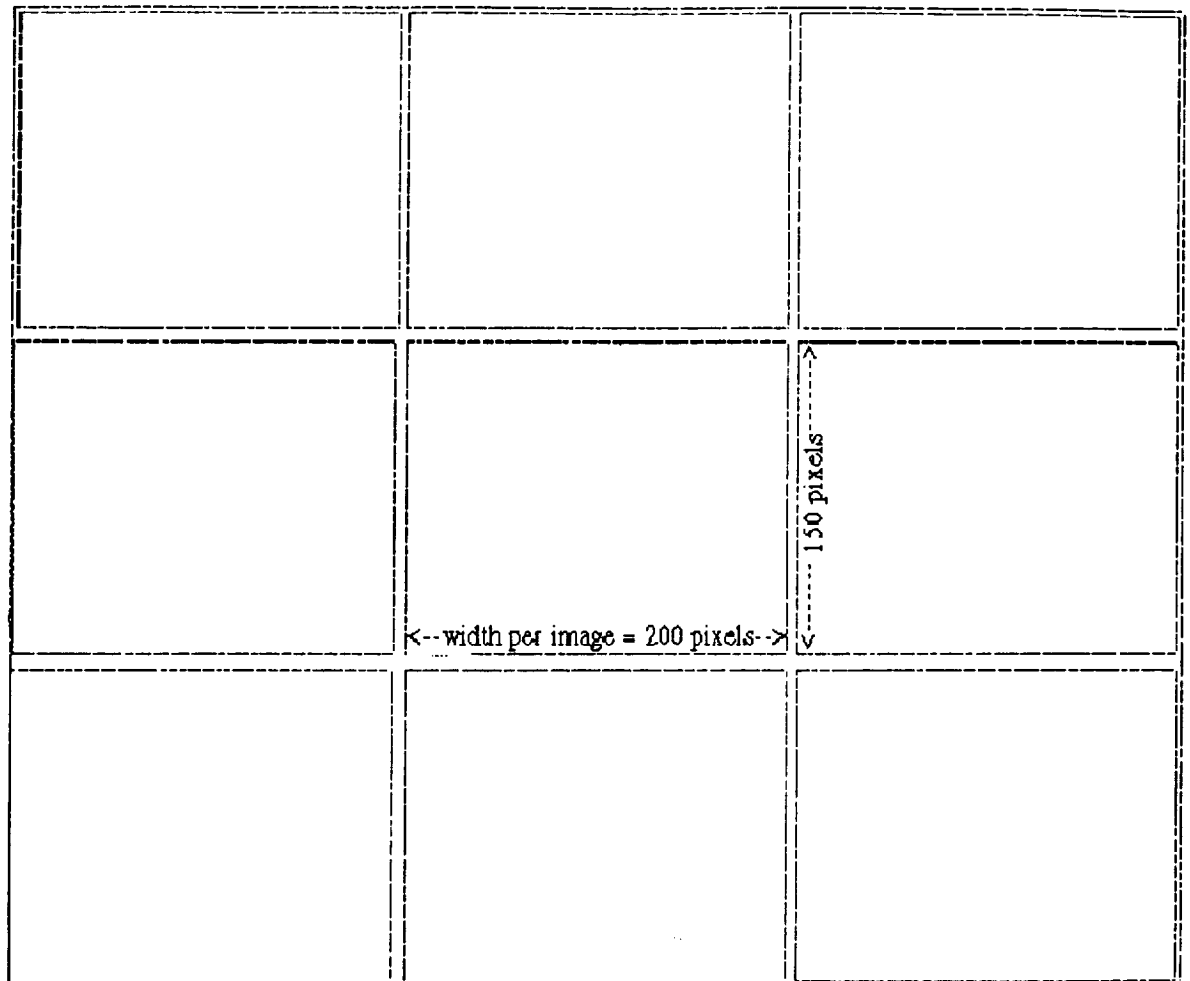
This is <http://science.nas.nasa.gov/people/walatka/WebToons/>

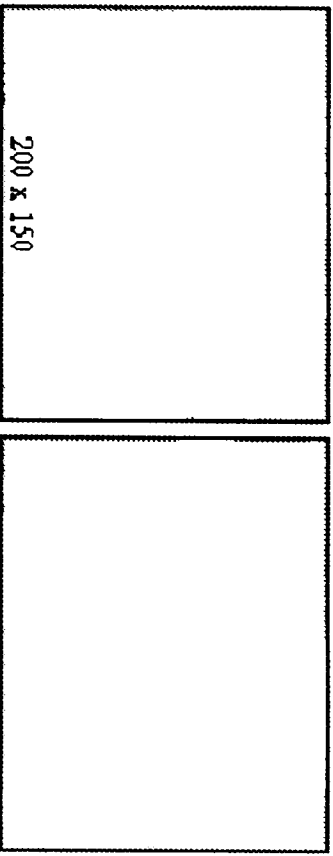


Updated: October 29, 1998
WebWork: Pam Walatka
NASA Responsible official:
Lisa Reid

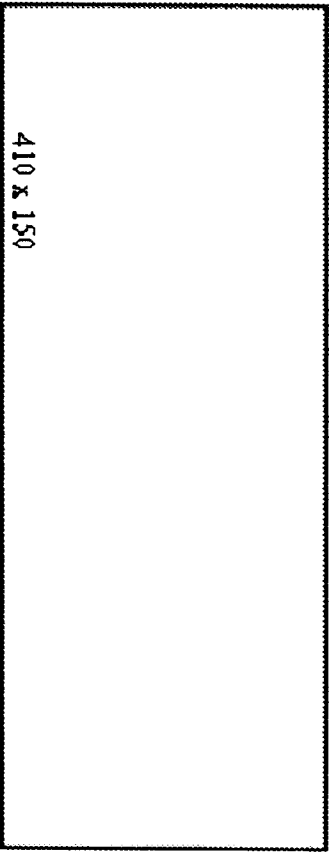
Appendix A

Sample Storyboards

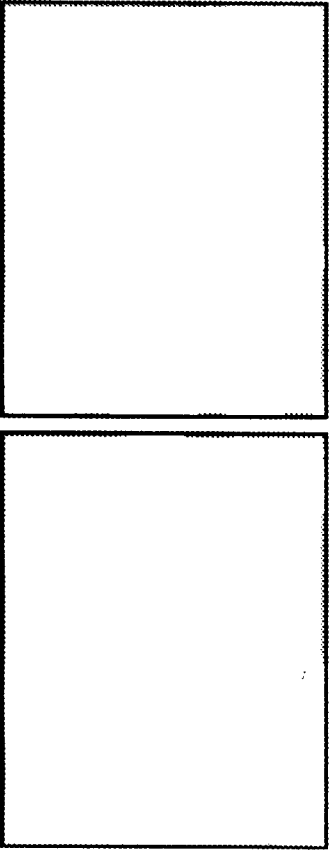


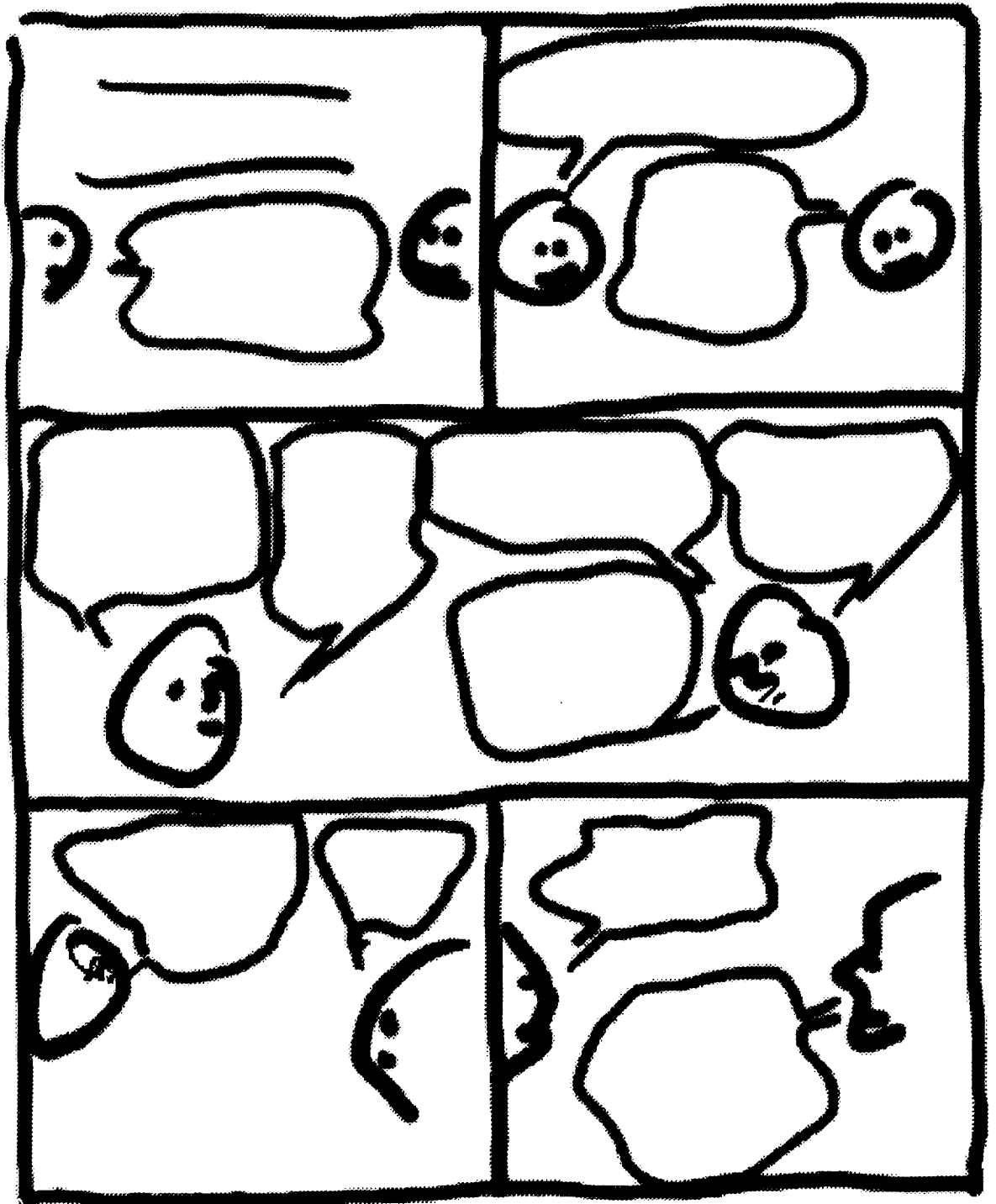


200 x 150



410 x 150





Appendix B

Map Coordinates Quiz Answer

```
<area shape="rect" coords="213,0,365,164"  
href="chapter3.html#section2">
```


Appendix C

Sample HTML for Imagemap Tables for WebToons

```
<!-- ***TABLE of IMAGEMAPS for WEBTOONS***-->
<table border width="414" >
<tr> <!-- Top Row-->
<td>
<!-- ***Toon 1***-->
    <map name="1toon.map">
        <area shape="rect" coords="0,0,200,68" href="index.html">
        <area shape="rect" coords="0,69,200,130" href="chapter1.html">
        <area shape="rect" coords="0,131,200,150" href=" ../home.html">
    </map>
    
</td>
<td>
<!-- ***Toon 2***-->
    <map name="2toon.map">
        <area shape="rect" coords="0,0,200,68" href="chapter2.html">
        <area shape="rect" coords="0,69,200,150" href="chapter2.html#photo">
    </map>
    
</td>
</tr> <!-- End Top Row-->

<tr> <!-- Middle Row = all one toon-->
<td colspan="2">
<!-- ***Toon 3***-->
    <map name="3toon.map">
        <area shape="rect" coords="0,0,107,88" href="chapter3.html#getin">
        <area shape="rect" coords="108,0,182,107" href="chapter3.html#pick">
        <area shape="rect" coords="183,0,305,63" href="chapter3.html#enhance">
        <area shape="rect" coords="306,0,410,79" href="chapter3.html#words">
        <area shape="rect" coords="87,64,308,138" href="chapter3.html#balloon">
        <area shape="default" href="chapter3.html">
    </map>
    
</td>
```

```

</tr> <!-- End Middle Row-->

<tr> <!-- Bottom Row-->
<td>
<!-- ***Toon 4***-->
  <!-- this one is just a linked image, not an imagemap-->
  <a href="chapter4.html"> </a>
</td>
<td>
<!-- ***Toon 5***-->
  <map name="5toon.map">
    <area shape="rect"    coords="0,0,200,64"    href="chapter5.html">
    <area shape="poly"    coords="70,70,150,70,170,90,170,130
      150,130 70,140 50,130 50,90,70,70" href="chapter5.html#html">
    <area shape="default"                                href="chapter5.html">
  </map>
  
</td>
</tr> <!-- End Bottom Row-->
</table>
<!-- End WEBTOON-->

```